Internet Engineering Task Force
INTERNET DRAFT
draft-ietf-tcpimpl-newreno-02.txt

Sally Floyd
ACIRI
Tom Henderson
U.C. Berkeley
February 1999
Expires: August 1999

The NewReno Modification to TCP's Fast Recovery Algorithm

Status of this Memo

Abstract

RFC 2001 [RFC2001] documents the following four intertwined TCP congestion control algorithms: Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery. RFC 2001-bis [RFC2001-bis] explicitly allows certain modifications of these algorithms, including modifications that use the TCP Selective Acknowledgement (SACK) option [MMFR96], and modifications that respond to ``partial acknowledgments'' (ACKs which cover new data, but not all the data outstanding when loss was detected) in the absence of SACK. This document describes a specific algorithm for responding to partial acknowledgments, referred to as NewReno. This response to partial acknowledgments was first proposed by Janey Hoe in [Hoe95].

1. Introduction

For the typical implementation of the TCP Fast Recovery algorithm described in [RFC2001-bis] (first implemented in the 1990 BSD Reno release, and referred to as the Reno algorithm in [FF96]), the TCP data sender only retransmits a packet after a retransmit timeout has occurred, or after three duplicate acknowledgements have arrived triggering the Fast Retransmit algorithm. A single retransmit timeout might result in the retransmission of several data packets, but each invocation of the Reno Fast Retransmit algorithm leads to the retransmission of only a single data packet.

Problems can arise, therefore, when multiple packets have been dropped from a single window of data and the Fast Retransmit and Fast Recovery algorithms are invoked. In this case, if the SACK option is available, the TCP sender has the information to make intelligent decisions about which packets to retransmit and which packets not to retransmit during Fast Recovery. This document applies only for TCP connections that are unable to use the TCP Selective Acknowledgement (SACK) option.

In the absence of SACK, there is little information available to the TCP sender in making retransmission decisions during Fast Recovery. From the three duplicate acknowledgements, the sender infers a packet loss, and retransmits the indicated packet. After this, the data sender could receive additional duplicate acknowledgements, as the data receiver acknowledges additional data packets that were already in flight when the sender entered Fast Retransmit.

In the case of multiple packets dropped from a single window of data, the first new information available to the sender comes when the sender receives an acknowledgement for the retransmitted packet (that is the packet retransmitted when Fast Retransmit was first entered). If there had been a single packet drop, then the acknowledgement for this packet will acknowledge all of the packets transmitted before Fast Retransmit was entered (in the absence of reordering). However, when there were multiple packet drops, then the acknowledgement for the retransmitted packet will acknowledge some but not all of the packets transmitted before the Fast Retransmit. We call this packet a partial acknowledgment.

Along with several other suggestions, [Hoe95] suggested that during Fast Recovery the TCP data sender respond to a partial acknowledgment by inferring that the indicated packet has been lost, and retransmitting that packet. This document describes a modification to the Fast Recovery algorithm in Reno TCP that incorporates a response to partial acknowledgements received during Fast Recovery. We call this modified Fast Recovery algorithm NewReno, because it is

a slight but significant variation of the basic Reno algorithm. This document does not discuss the other suggestions in [Hoe95] and [Hoe96], such as a change to the ssthresh parameter during Slow-Start, or the proposal to send a new packet for every two duplicate acknowledgements during Fast Recovery. The version of NewReno in this document also draws on other discussions of NewReno in the literature [LM97].

We do not claim that the NewReno version of Fast Recovery described here is an optimal modification of Fast Recovery for responding to partial acknowledgements, for TCPs that are unable to use SACK. Based on our experiences with the NewReno modification in the NS simulator [NS], we believe that this modification improves the performance of the Fast Retransmit and Fast Recovery algorithms in a wide variety of scenarios, and we are simply documenting it for the benefit of the IETF community. We encourage the use of this modification to Fast Recovery, and we further encourage feedback about operational experiences with this or related modifications.

2. Definitions

This document assumes that the reader is familiar with the terms MAXIMUM SEGMENT SIZE (MSS), CONGESTION WINDOW (cwnd), and FLIGHT SIZE (FlightSize) defined in [RFC2001-bis]. FLIGHT SIZE is defined as in [RFC2001-bis] as follows:

> FLIGHT SIZE:
> The amount of data that has been sent but not yet acknowledged.

3. The Fast Retransmit and Fast Recovery algorithms in NewReno

The standard implementation of the Fast Retransmit and Fast Recovery algorithms is given in [RFC2001-bis]. The NewReno modification of these algorithms is given below. This NewReno modification differs from the implementation in [RFC2001-bis] only in the introduction of the variable "recover" in step 1, and in the response to a partial or new acknowledgement in step 5.

1. When the third duplicate ACK is received, set ssthresh to no more than the value given in equation 1 below. (This is equation 3 from [RFC2001-bis]).

   ssthresh = max (FlightSize / 2, 2*MSS) (1)

   Record the highest sequence number transmitted in the variable "recover".

2. Retransmit the lost segment and set cwnd to ssthresh plus 3*MSS. This artificially "inflates" the congestion window by the number of segments (three) that have left the network and which the receiver has buffered.

3. For each additional duplicate ACK received, increment cwnd by MSS. This artificially inflates the congestion window in order to reflect the additional segment that has left the network.

4. Transmit a segment, if allowed by the new value of cwnd and the receiver's advertised window.

5. When an ACK arrives that acknowledges new data, this ACK could be the acknowledgment elicited by the retransmission from step 2, or elicited by a later retransmission.

   If this ACK acknowledges all of the data up to and including "recover", then the ACK acknowledges all the intermediate segments sent between the original transmission of the lost segment and the receipt of the third duplicate ACK. Set cwnd to either (1) min (ssthresh, FlightSize + MSS); or (2) ssthresh, where ssthresh is the value set in step 1; this is termed "deflating" the window. (We note that "FlightSize" in step 1 referred to the amount of data outstanding in step 1, when Fast Recovery was entered, while "FlightSize" in step 5 refers to the amount of data outstanding in step 5, when Fast Recovery is exited.) If the second option is selected, the implementation should take measures to avoid a possible burst of data, in case the amount of data outstanding in the network was much less than the new congestion window allows [HTH98]. Clear the counter recording the number of duplicate acknowledgements, exiting the Fast Recovery procedure.

   If this ACK does *not* acknowledge all of the data up to and including "recover", then this is a partial ACK. In this case, retransmit the first unacknowledged segment. Deflate the congestion window by the amount of new data acknowledged, then add back one MSS and send a new segment if permitted by the new value of cwnd. This "partial window deflation" attempts to ensure that, when Fast Recovery eventually ends, approximately ssthresh amount of data will be outstanding in the network. Do not clear the counter recording the number of duplicate acknowledgements (i.e., do not exit the Fast Recovery procedure).

   For the first partial ACK that arrives during Fast Recovery, also reset the retransmit timer.

Note that in Step 5, the congestion window is deflated when a partial

acknowledgement is received. The congestion window was likely to have been inflated considerably when the partial acknowledgement was received. In addition, depending on the original pattern of packet losses, the partial acknowledgement might acknowledge nearly a window of data. In this case, if the congestion window was not deflated, the data sender might be able to send nearly a window of data back-to-back.

There are several possible variants to the simple response to partial acknowledgements described above. First, there is a question of when to reset the retransmit timer after a partial acknowledgement. This is discussed further in Section 4 below.

There is a related question of how many packets to retransmit after each partial acknowledgement. The algorithm described above retransmits a single packet after each partial acknowledgement. This is the most conservative alternative, in that it is the least likely to result in an unnecessarily-retransmitted packet. A variant that would recover faster from a window with many packet drops would be to effectively Slow-Start, requiring less than N roundtrip times to recover from N losses [Hoe96]. With this slightly-more-aggressive response to partial acknowledgements, it would be advantageous to reset the retransmit timer after each retransmission. Because we have not experimented with this variant in our simulator, we do not discuss this variant further in this document.

A third question involves avoiding multiple Fast Retransmits caused by the retransmission of packets already received by the receiver. This is discussed in Section 5 below. Avoiding multiple Fast Retransmits is particularly important if more aggressive responses to partial acknowledgements are implemented, because in this case the sender is more likely to retransmit packets already received by the receiver.

As a final note, we would observe that in the absence of the SACK option, the data sender is working from limited information. One could spend a great deal of time considering exactly which variant of Fast Recovery is optimal for which scenario in this case. When the issue of recovery from multiple dropped packets from a single window of data is of particular importance, the best alternative would be to use the SACK option.

4. Resetting the retransmit timer.

The algorithm in Section 3 resets the retransmit timer only after the first partial ACK. In this case, if a large number of packets were dropped from a window of data, the TCP data sender's retransmit timer will ultimately expire, and the TCP data sender will invoke Slow-

Start. (This is illustrated on page 12 of [F98].) We call this the Impatient variant of NewReno.

In contrast, the NewReno simulations in [FF96] illustrate the algorithm described above, with the modification that the retransmit timer is reset after each partial acknowledgement. We call this the Slow-but-Steady variant of NewReno. In this case, for a window with a large number of packet drops, the TCP data sender retransmits at most one packet per roundtrip time. (This behavior is illustrated in the New-Reno TCP simulation of Figure 5 in [FF96], and on page 11 of [F98].)

For TCP implementations where the Retransmission Timeout Value (RTO) is generally not much larger than the round-trip time (RTT), the Impatient variant can result in a retransmit timeout even in a scenario with a small number of packet drops. For TCP implementations where the Retransmission Timeout Value (RTO) is usually considerably larger than the round-trip time (RTT), the Slow-but-Steady variant can remain in Fast Recovery for a long time when multiple packets have been dropped from a window of data. Neither of these variants are optimal; one possibility for a more optimal algorithm might be one that recovered more quickly from multiple packet drops, and combined this with the Slow-but-Steady variant in terms of resetting the retransmit timers. We note, however, that there is a limitation to the potential performance in this case in the absence of the SACK option.

5. Avoiding Multiple Fast Retransmits

In the absence of the SACK option, a duplicate acknowledgement carries no information to identify the data packet or packets at the TCP data receiver that triggered that duplicate acknowledgement. The TCP data sender is unable to distinguish between a duplicate acknowledgement that results from a lost or delayed data packet, and a duplicate acknowledgement that results from the sender's retransmission of a data packet that had already been received at the TCP data receiver. Because of this, multiple segment losses from a single window of data can sometimes result in unnecessary multiple Fast Retransmits (and multiple reductions of the congestion window) [Flo94].

With the Reno or NewReno Fast Retransmit and Fast Recovery algorithms, the performance problems caused by multiple Fast Retransmits are relatively minor (compared to the potential problems with Tahoe TCP, which does not implement Fast Recovery). Nevertheless, these unnecessary Fast Retransmits can occur with Reno or NewReno TCP, particularly if a Retransmit Timeout occurs during Fast Recovery. (This is illustrated for Reno on page 6 of [F98], and

for NewReno on page 8 of [F98].) With NewReno, the data sender remains in Fast Recovery until either a Retransmit Timeout, or until all of the data outstanding when Fast Retransmit was entered has been acknowledged. Thus with NewReno, the problem of multiple Fast Retransmits from a single window of data can only occur after a Retransmit Timeout.

The following modification to the algorithms in Section 3 eliminates the problem of multiple Fast Retransmits. (This modification is called "bugfix" in [F98], and is illustrated on pages 7 and 9.) This modification uses a new variable "send_high", whose initial value is zero. After each retransmit timeout, the highest sequence number transmitted so far is recorded in the variable "send_high".

If, after a retransmit timeout, the TCP data sender retransmits three consecutive packets that have already been received by the data receiver, then the TCP data sender will receive three duplicate acknowledgements that do not acknowledge "send_high". In this case, the duplicate acknowledgements are not an indication of a new instance of congestion. They are simply an indication that the sender has unnecessarily retransmitted at least three packets.

We note that if the TCP data sender receives three duplicate acknowledgements that do not acknowledge "send_high", the sender does not know whether these duplicate acknowledgements resulted from a new packet drop or not. For a TCP that implements the bugfix described in this section for avoiding multiple fast retransmits, the sender does not infer a packet drop from duplicate acknowledgements in these circumstances. As always, the retransmit timer is the backup mechanism for inferring packet loss in this case.

The modification to Fast Retransmit for avoiding multiple Fast Retransmits replaces Step 1 in Section 3 with Step 1A below. In addition, the modification adds Step 6 below:

1A. When the third duplicate ACK is received, check to see if those duplicate ACKs cover more than "send_high". If they do, then set ssthresh to no more than the value given in equation 1, record the highest sequence number transmitted in the variables "recover" and "send_high", and go to Step 2. If the duplicate ACKs don't cover send_high, then do nothing. That is, do not enter the Fast Retransmit and Fast Recovery procedure, do not change ssthresh, and do not go to Step 2 to retransmit the "lost" segment.

Steps 2-5 are the same as those steps in Section 3 above.

6. After a retransmit timeout, record the highest sequence number

transmitted in the variable "send_high". Do not change the variable "recover".

Step 1A above, in checking whether the duplicate ACKs cover *more* than "send_high", is the Careful variant of this algorithm. Another possible variant would be to require simply that the three duplicate acknowledgements *cover* "send_high" before initiating another Fast Retransmit. We call this the Less Careful variant to Fast Retransmit.

There are two separate scenarios in which the TCP sender could receive three duplicate acknowledgements acknowledging "send_high" but no more than "send_high". One scenario would be that the data sender transmitted four packets with sequence numbers higher than "send_high", that the first packet was dropped in the network, and the following three packets triggered three duplicate acknowledgements acknowledging "send_high". The second scenario would be that the sender unnecessarily retransmitted three packets below "send_high", and that these three packets triggered three duplicate acknowledgements acknowledging "send_high". In the absence of SACK, the TCP sender in unable to distinguish between these two scenarios.

For the Careful variant of Fast Retransmit, the data sender would have to wait for a retransmit timeout in the first scenario, but would not have an unnecessary Fast Retransmit in the second scenario. For the Less Careful variant to Fast Retransmit, the data sender would Fast Retransmit as desired in the first scenario, and would unnecessarily Fast Retransmit in the second scenario. The NS simulator has implemented the Less Careful variant of NewReno, and the TCP implementation in Sun's Solaris 7 implements the Careful variant. This document recommends the Careful variant given in Step 1A above.

6. Implementation issues for the data receiver.

[RFC2001] specifies that "Out-of-order data segments SHOULD be acknowledged immediately, in order to trigger the fast retransmit algorithm." Neal Cardwell has noted [C98] that some data receivers do not send an immediate acknowledgement when they send a partial acknowledgment, but instead wait first for their delayed acknowledgement timer to expire. As [C98] notes, this severely limits the potential benefit from NewReno by delaying the receipt of the partial acknowledgement at the data sender. Our recommendation is that the data receiver send an immediate acknowledgement for an out-of-order segment, even when that out-of-order segment fills a hole in the buffer.

7. Simulations

Simulations with NewReno are illustrated with the validation test "tcl/test/test-all-newreno" in the NS simulator. The command "../../ns test-suite-newreno.tcl reno" shows a simulation with Reno TCP, illustrating the data sender's lack of response to a partial acknowledgement. In contrast, the command "../../ns test-suite-newreno.tcl newreno_B" shows a simulation with the same scenario using the NewReno algorithms described in this paper.

The tests "../../ns test-suite-newreno.tcl newreno1_B0" and "../../ns test-suite-newreno.tcl newreno1_B" show the Slow-but-Steady and the Impatient variants of NewReno, respectively.

8. Conclusions

Our recommendation is that TCP implementations include the NewReno modification to the Fast Recovery algorithm given in Section 3, along with the modification for avoiding multiple Fast Retransmits given in Section 5. The NewReno modification given in Section 3 can be important even for TCP implementations that support the SACK option, because the SACK option can only be used for TCP connections when both TCP end-nodes support the SACK option. The NewReno modification given in Section 3 implements the Impatient rather than the Slow-but-Steady variant of NewReno.

While this document mentions several possible variations to the NewReno algorithm, we have not explored all of these possible variations, and therefore are unable to make recommendations about some of them. Our belief is that the differences between any two variants of NewReno are small compared to the differences between Reno and NewReno. That is, the important thing is to implement NewReno instead of Reno, for a TCP invocation without SACK; it is less important exactly *which* variant of NewReno is implemented.

9. Acknowledgements

Many thanks to Mark Allman, Vern Paxson, Kacheong Poon, and Bernie Volz for detailed feedback on this document.

10. References

[C98] Neal Cardwell, "delayed ACKs for retransmitted packets: ouch!". November 1998. Email to the tcpimpl mailing list, Message-ID "Pine.LNX.4.02A.9811021421340.26785-100000@sake.cs.washington.edu", archived at "http://tcp-impl.lerc.nasa.gov/tcp-impl".

[F98] Sally Floyd. Revisions to RFC 2001. Presentation to the TCPIMPL Working Group, August 1998. URLs "ftp://ftp.ee.lbl.gov/talks/sf-tcpimpl-aug98.ps" and "ftp://ftp.ee.lbl.gov/talks/sf-tcpimpl-aug98.pdf".

[FF96] Kevin Fall and Sally Floyd. Simulation-based Comparisons of Tahoe, Reno and SACK TCP. Computer Communication Review, July 1996. URL "ftp://ftp.ee.lbl.gov/papers/sacks.ps.Z".

[Flo94] S. Floyd, TCP and Successive Fast Retransmits. Technical report, October 1994. URL "ftp://ftp.ee.lbl.gov/papers/fastretrans.ps".

[Hen98] Tom Henderson, Re: NewReno and the 2001 Revision. September 1998. Email to the tcpimpl mailing list, Message ID "Pine.BSI.3.95.980923224136.26134A-100000@raptor.CS.Berkeley.EDU", archived at "http://tcp-impl.lerc.nasa.gov/tcp-impl".

[Hoe95] J. Hoe, Startup Dynamics of TCP's Congestion Control and Avoidance Schemes. Master's Thesis, MIT, 1995. URL "http://ana-www.lcs.mit.edu/anaweb/ps-papers/hoe-thesis.ps".

[Hoe96] J. Hoe, Improving the Start-up Behavior of a Congestion Control Scheme for TCP. In ACM SIGCOMM, August 1996. URL "http://www.acm.org/sigcomm/sigcomm96/program.html".

[HTH98] A. Hughes, J. Touch, J. Heidemann, Issues in TCP Slow-Start Restart After Idle, Work in progress, March 1998.

[LM97] Dong Lin and Robert Morris, "Dynamics of Random Early Detection", SIGCOMM 97, September 1997. URL "http://www.acm.org/sigcomm/sigcomm97/program.html".

[MMFR96] M. Mathis, J. Mahdavi, S. Floyd, A. Romanow, "TCP Selective Acknowledgement Options", RFC 2018, October 1996.

[NS] The UCB/LBNL/VINT Network Simulator (NS). URL "http://www-mash.cs.berkeley.edu/ns/".

[RFC2001] W. Stevens, "TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms", RFC 2001, January 1997.

[RFC2001-bis] W. Stevens, M. Allman, and V. Paxson, "TCP Congestion Control", draft-ietf-tcpimpl-cong-control-00.txt, August 1998.

11. Security Considerations

RFC 2001-bis discusses general security considerations concerning TCP congestion control. This document describes a specific algorithm that conforms with the congestion control requirements of RFC 2001-bis, and so those considerations apply to this algorithm, too. There are no known additional security concerns for this specific algorithm.

AUTHORS' ADDRESSES

Sally Floyd
AT&T Center for Internet Research at ICSI (ACIRI)
Phone: +1 (510) 642-4274 x189
Email: floyd@acm.org
URL: http://www-nrg.ee.lbl.gov/floyd/

Tom Henderson
University of California at Berkeley
Phone: +1 (510) 642-8919
Email: tomh@cs.berkeley.edu
URL: http://www.cs.berkeley.edu/~tomh/

This draft was created in February 1999.
It expires August 1999.